Advanced Software Fault Tolerance Strategies for Mission Critical Spacecraft Applications

for
NASA Ames Research Center IV&V Facility
Software Initiative UPN 323-08

Task 3 Interim Report
September 30, 1999
Assessment of X-38 201 Software Architecture and
Software Development Approach

Prepared by JSC NX Technology Division

Richard Gilreath
Paul Porter
Charles Nagy

Approved by M. Himel/Chief, NX Technology Division

1 Purpose

This report, the third in a series of four reports, examines software fault tolerance and reliability methods in the evolving X-38 201 flight software. The purpose is to provide concrete examples of methods discussed in the first report. Beyond simply identifying those methods currently being applied in the software, this report *suggests* opportunities for applying other methods and enhancing those methods currently being used to improve safety and reliability in the X-38 software.

By extension, the methods discussed in this report could also be applied in other NASA programs using flight software. (Because these methods are not unique to flight software, they have applicability to other kinds of systems, as well.) In the context of the research objective, to identify software fault tolerance and reliability strategies, the X-38 software serves only as a model. X-38 software product assurance is *not* the objective of this research effort or this report. Any improvement in the X-38 software product that may result from the suggestions offered will simply be a byproduct of this effort.

2 Background

2.1 Previous Findings

The first report surveyed methods used to provide software fault tolerant and extremely reliable systems. Findings from the first report are summarized here for the convenience of the reader and to establish a context for the remaining discussion:

1. "N-version and recovery form the basis of all software fault tolerant systems."
2. "Formal methods show great promise but are hampered by several drawbacks…"
3. "The use of assertions can be quite powerful and is within the grasp of almost all developers."
4. "Model state checking is a strong approach to identifying incomplete requirements."
5. "The real world will require a combination of approaches."

With regard to the second finding, one of the drawbacks identified was the inability of formal methods to identify incomplete (i.e., omitted) requirements. This is a challenge to software development that is not unique to formal methods.

An example was provided for the final finding, suggesting an approach to developing software fault tolerant systems that includes the use of N-version development for critical functions in combination with other methods.

2.2 Definitions

To establish a context for the discussion to follow, it is important to define several key concepts. Heimerdinger and Weinstock [1] offer a conceptual framework for discussing fault tolerance that will be generally adopted in this report. They define the *failure* of a system as "the deviation of the service delivered by a system from the system specification" and a *fault* as the "adjudged cause of a failure." Note that this definition assumes a *valid* system specification – i.e., one that is correct with regard to specifying the intended service. Further, a fault is itself a failure in a component of the system, for example, in a hardware or software component (or subsystem), or at an interface with something external to the system, for example, with a user or some other system providing an input to the system.

Particularly, Heimerdinger and Weinstock identify four approaches to developing *dependable* systems, briefly summarized here in the context of software fault tolerance. *Fault avoidance* and *fault removal* involve techniques to eliminate faults, in design and testing, respectively. These techniques are aimed at developing software that is fault-free before it is operational (with the added benefit of reducing the cost of software maintenance). *Fault tolerance* and *fault evasion* involve techniques applied *at execution time* to prevent software faults from causing a system failure. Fault tolerance involves detecting faults and mitigating their effects with appropriate actions such as masking, containment, or recovery. Fault evasion

[1] See [1].

involves predicting the occurrence of a fault based on some observed behavior in the system (*implying the that some other related fault is manifesting as symptoms in the observed behavior)* and taking some action to mitigate the possible effects of an anticipated fault before it can result in system failure.

Techniques to assure the *validity* (i.e., the correctness) of the system specification might be added to the fault avoidance techniques, extending the definition of failure to include failing to perform the *intended or desired* function of the system. Particularly, this would include safety requirements, whether formally specified or not. Thus, when considering methods to improve software reliability, it is important to also consider the validity of the system specification, particularly with regard to software safety. Hatton[2] discusses the distinction between reliability and safety, generally associating reliability with consistent behavior, and safety with correct behavior.

Several additional concepts pertinent to the discussion are defined in the brief overview of the X-38 system architecture provided in this report.

2.3 Software Fault Tolerance and Reliability Methods

The first report identified a number of software fault tolerance and reliability methods. Table 1 identifies those methods pertinent to the discussion in this report, including a refined subset of the methods identified in the first report, as well as some additional ones. Further, these are categorized according to the approaches described in the previous section.

Note that hardware redundancy is not listed simply because it is assumed to underlay many of the software fault tolerance and fault evasion methods that would make recovery from certain faults possible. Also note that this table excludes methods that might be applied outside the software system boundary, for example, operational scenarios to respond to a total failure of the system.

[2] See [2], p11.

Table 1 -- Fault Prevention and Handling Methods

Method	Prevent Faults		Deal with Faults	
	Fault Avoidance	**Fault Removal**	**Fault Tolerance**	**Fault Evasion**
N-versioning	X		X	
Formal Methods[1]	X			
State Modeling[2]	X			
Reliability Modeling[3]	X			
Fault Analysis[4]	X			
Fault Injection		X		
Partitioning	X		X	
Voting			X	
Retry			X	
Recovery Block[5]			X	
Distributed Recovery Block			X	
Heartbeat Messages			X	
Watchdog Timers			X	
Input Analysis				X
Output Analysis				X
Consistency Checks				X
Subsetting				
Design Paradigm	X			
Coding Standards	X			
Environment Standards	X			
Automated Tools				
Requirements Management	X			
Configuration Management	X			
Design	X			
Code Analysis		X		
Code Generation		X		
Debug		X		
Test Generation		X		
Test Effectiveness Analysis		X		

[1] Also see Interim Report 1 and p.10, [10].
[2] See Interim Report 1.
[3] See p.9, [10].
[4] See Appendix B,[11]
[5] Recovery blocks are distinguished from distributed recovery blocks in this list. The former are confined in scope to a single processor; the latter involve communication among multiple processors. In addition, checkpointing methods are generally considered variations of recovery block methods; thus, they are not listed separately.

The following paragraphs expand on the definitions provided in the first report, beginning with a brief discussion of redundancy management.

Redundancy management, the "cornerstone" of fault tolerant systems, focuses on managing redundant hardware. Unlike hardware, which can break physically (e.g., through deterioration or exposure to environmental conditions), software (assuming a properly functioning host platform) is either correct or not – it cannot "break" *(though there can be latent faults which manifest under the requisite conditions).* When considering software fault tolerance, the concept of software redundancy is meaningless unless functional redundancy is implemented in non-identical software (i.e., N-versions of the software). Simple *physical* redundancy of software (i.e., replication of the software) accomplishes nothing in the absence of hardware faults. A logic or design error in software causing a fault (data or timing) would be replicated in all redundant copies.

N-versioning involves methods to provide *functional* redundancy in independently implemented versions of software (i.e., in functionally, but not physically redundant code). In this sense, it is a fault tolerance method, which has at least two common connotations. Leveson offers the following definition:

> "An N-version system attempts to incorporate fault tolerance into software by executing multiple versions of a program that have been prepared independently. Their outputs are collected and examined by a decision function that chooses the output to be used by the system. For example, if the outputs are not identical but should be, the decision function might choose the majority value if there is one."[3]

This connotation involves a "decision function", that is, some voting mechanism. The second connotation eliminates the decision function, executing the multiple versions *sequentially*. For example, a 2-version system would consist of an active primary version and a dormant backup version, which would become active only when the primary version failed. In a "pure" N-version approach both the primary and backup versions of the system, being functionally identical, would deliver the *same* service. An approach which involves an independently developed backup version of software which delivers a different or degraded level of service (and is, therefore, not functionally identical to the primary version) simply uses *different* systems, and can be described as a *primary-backup* approach.

N-versioning can be applied at various levels in the software, not only at the system level. For example, multiple versions of a software component that implements a critical algorithm could be developed independently and executed simultaneously in conjunction with a decision function, or used sequentially in a primary-backup mode, to provide tolerance of faulty output values.

N-versioning also connotes an *approach* taken in developing software. In this sense, it is a fault avoidance method, which can be applied at various stages of software development. For example, at one extreme, multiple versions of a software system could be developed from a common system specification that defines the service to be delivered by the system, possibly varying the processor platform or the design paradigm. At the other extreme, multiple versions of code could be developed from a low level functional requirements specification, perhaps varying the programming language or the algorithm used.

Implicit in an N-versioning approach is an assumption concerning human factors that independent efforts will yield different results. This is a controversial aspect of the approach and the subject of research and debate beyond the scope of this discussion. In any case, common sense dictates that an N-versioning approach increases the cost of the software, since, by definition, it involves N development efforts.

Partitioning involves methods to limit the scope of a software fault's effects to a particular software component or software subystem (or layer of software in the abstract sense). Partitioning includes methods *designed* into the software to contain software faults (and limit the scope of fault recovery) in a manner analogous to containing hardware faults. Further, partitioning can be supported by hardware, as well as software mechanisms, to enforce the partitioning and participate in fault detection and recovery. For

[3] See [3].

example, by partitioning an application into tasks (with hardware helping to protect the task memory and to limit the task's resource consumption), faults manifesting in one task may be prevented from affecting other tasks and the system. Implementing software in layers, tasks, modules, and procedures are among the techniques used to partition software. To reiterate, software partitioning must be considered in the design (as hardware fault containment must be considered in hardware design), so to this extent it is a fault prevention method. To the extent that this design is reflected in the implemented software's ability to handle faults, it can be considered a fault tolerance method.

Several methods can be categorized together as *comparison* methods. **Voting** involves comparing a set of input or output values from different sources (implying redundant sources) to select a single representative value, typically a majority value, an average, or an historically based value. **Input analysis** involves acceptance tests of input values (or state conditions) applied *before* executing some process. **Output analysis** involves acceptance tests applied to output values (or state conditions) *after* executing some process. In the event that an acceptance test fails, some recovery action is taken, for example, to guarantee that the value is an acceptable input or output value. The objective is to prevent propagation of a data fault beyond the process by guaranteeing the correctness (or the reasonableness) of the value. Note that an acceptance test can be a simple comparison or a more complex test, involving, for example, some kind of statistical analysis. Both input analysis and output analysis can be categorized as **assertions**, involving tests against specific process requirements. **Consistency checks** involve tests of data values at selected points in processing to provide early detection of corrupted values, and to permit appropriate action to contain and recover from the fault causing the corruption. For example, consistency checks could be applied to memory structures such as stacks or queues to verify that the structure has not been corrupted and that values in the structure are consistent.

Recovery blocks are distinguished from **distributed recovery blocks** in this list. Both methods simply involve maintaining data and state information that can be used to restore a faulting process to a previous non-faulty state following detection of a fault. Recovery blocks involve saving information on a single processor. Distributed recovery blocks involve maintaining copies of the recovery information in communicating processors. Note that **checkpointing**, which was identified in the first report, can be considered a recovery block (or distributed recovery block) method, typically connoting the use of disk storage to maintain recovery information.

Retry is a method commonly used in I/O interface software to recover from transient errors. This can be a simple repeated attempt of an I/O operation (e.g., to recover from a hardware "glitch") or it can be more elaborate, involving a self-correcting algorithm. For example, disk device driver software may retry unsuccessful I/O operations, varying commands on successive attempts to adjust alignment of the disk read-write heads.

Fault Injection is simply a testing method that involves systematically introducing faults into an executing software system to evaluate the system's ability to handle the faults. The objective is to measure a system's resilience to fault conditions, that is, its ability to tolerate or evade known faults, including data value and timing faults. To apply fault injection requires detailed knowledge of input and output data requirements (e.g., value ranges) and timing requirements, as well as intimate knowledge of the software structure and fault handling capabilities.

Fault Analysis includes primarily the **Failure Modes and Effects Analysis (FMEA)** and **Software Fault Tree Analysis (SFTA)** methods identified in the first report. FMEA examines the effects of software faults on a system in a "bottom up" approach, for example, predicting the effect on the system if a particular software component fails by producing an incorrect output value. SFTA examines the possible causes of system failure in a "top down" approach, examining the potential of various system components to cause a failure.

The objective of **"Subsetting"** is to constrain the software design or code – to organize it and to simplify it, making it more manageable and less prone to error. Thus, a **design paradigm** organizes the software into manageable units which may be inherently more error-free (e.g., in an object-oriented design paradigm). A **coding standard** constrains the code product to a language subset that is free of error-prone

language elements. The **programming environment** imposes implementation constraints to eliminate programming methods (e.g., software interface methods) that are likely to produce errors in the system.

The NASA Guidebook for Safety Critical Software defines Formal Methods as follows:[4]

1. "The use of formal logic, discrete mathematics, and machine-readable languages to specify and verify software.
2. The use of mathematical techniques in design and analysis."

Note that definitions for other methods listed are presented in the first report and in the references cited in the footnotes in Table 1.

2.4 X-38 Project Overview

In a nutshell, the X-38 201 vehicle is a prototype "lifeboat" for the crew of the International Space Station. The following paragraph, extracted from the X-38 Software Development Plan, provides an overview of the X-38 201 project and briefly summarizes some major requirements:

> "The X-38 201 Vehicle is a space prototype vehicle, which will serve as a proof of concept for the Crew Return Vehicle (CRV) for the International Space Station (ISS). The intent of the X-38 201 vehicle is to make the vehicle and its tests as CRV-like, as possible. There will, however, be several differences since the 201 vehicle will be released from the Shuttle, will not be manned, and only has to meet a one fault tolerance requirement both inside and outside the cabin."

The X-38 201 vehicle architecture consists of a number of flight critical (sub)systems, including a guidance, navigation and control system (with Global Positioning System interfaces), a power control system, a life support system, a parafoil landing system, and a data communications system which supports telemetry data downlink and remote commanding from various sources. Central to all of these is a multi-processor computer system which manages all vehicle systems. While flight-critical in the X-38 201 prototype vehicle, these subsystems would be safety-critical in the X-38 Crew Return Vehicle.

2.5 Approach to X-38 201 Software Assessment

The X-38 201 software development effort is in the early stages of an evolutionary development with only preliminary drafts of some major documents available. Given the preliminary state of the documentation, a short review time (which would not permit lengthy discussion with developers), and the necessity to avoid interfering with the development effort, the assessment presented in this report is based primarily on the documentation snapshot. The draft documents that were reviewed include the following (which are further identified in the bibliography):

- Software Architecture Definition Document
- Software Requirements Specifications (SRS's) for key components
- Software Interface Control Documents (ICD's) (some)
- Software Verification and Validation Plan
- Software Development Plan
- Fault Tolerant System Services (FTSS) Applications Programming Interface (API) Specification (FTSS will be discussed in the overview of the X-38 architecture)

Given the state of the documentation and the level of the review, there obviously are a number of unanswered questions. As well, there may be suggestions in this report that have already been considered by the project, but which are simply not reflected in the reviewed documentation.

[4] See p.A-12 in [11].

3 Assessment of X-38 201 Software Architecture

3.1 Flight Critical Computer System Overview

The X-38 201 flight critical computer system monitors and controls all onboard vehicle subsystems to control the vehicle through all flight phases. The primary software functions include acquiring sensor information from onboard subsystems and processing that information to generate commands to control the subsystems. As stated in one of the requirements specifications, the system "… is also responsible for monitoring overall health and status of all subsystems *and maintaining requisite functionality in the presence of faults.*"[5] In addition, the software is required to collect and downlink telemetry data, including sensor and control information, and to receive and process commands from various remote sources that can exert external control of the vehicle subsystems.

Among the major system requirements[6]:

1. The "architecture" is required to be 1 fault tolerant ("with 2 fault tolerance desirable in flight critical systems").
2. The system is required to have four fault containment regions with "critical sensors, critical effectors, and power systems … distributed such that two failures do not bring down the entire vehicle."
3. The software is required to operate in three rate groups – 50 Hz, 10 Hz, and 1 Hz (20, 100, and 1000 millisecond cycles, respectively), with the "50 and 10 Hz rate groups … divided into flight critical and non-flight critical rate groups."
4. The architecture is driven by "two main requirements":
   - "All 50 Hz flight critical tasks (from sensor read to effector write) must meet a 10 ms transport lag."
   - "All 10 Hz flight critical tasks (from sensor read to effector write) must meet a 50 ms transport lag."

Following is a brief overview of the hardware architecture, which also defines some key concepts presented in the X-38 software documentation.[7] Figure 1 illustrates the major hardware components of the system. The system is characterized as an "instantiation" of Draper Laboratory's[8] Fault Tolerant Parallel Processor (FTPP) architecture. Central to the architecture of this system are five interconnected Flight Critical (computer) Chassis (FCC's), four of which are configured identically on a VME backplane (a 64-bit bus) with two processors, a Network Element (NE) card, and an array of I/O interface cards. Each of the identically configured FCC's is a Fault Containment Region (FCR). The two processors in each chassis are a Flight Critical Processor (FCP) and an Instrumentation Control Processor (ICP). The ICP serves as an I/O processor, managing and controlling the I/O interfaces to the vehicle subsystems. While the array of I/O interface cards in each FCR is identical, all ICP's do not interface with the same devices, though redundant interfaces to critical subsystems are provided. The FCP is the main processor in each FCC. It hosts the flight critical applications software. The FCP in each of the four FCR's also has limited I/O processing responsibility, supporting interfaces to redundant onboard Command and Telemetry Computer (CTC) systems, as well as a timer interface. Within an FCC, the FCP and ICP are connected to the NE which is the primary means for interprocessor communications, with some minimal communication via shared memory on the VME backplane and with a synchronization interrupt mechanism.

The fifth FCC is identical to the others with the exception of the I/O interface cards. Its only external I/O interface is to an X-38 simulator.[9] Thus, it cannot actively monitor or control any of the vehicle subsystems. To distinguish this FCC from the others, it is identified as a Network Element Fifth Unit (NEFU). It's primary function, to be explained more fully, is to participate in recovery in the event of a

[5] Section 2.1, p.6, FTPP SRS [6].
[6] See Slide 8, PDR Material [7].
[7] See SADD [5] and FTPP SRS [6], the primary sources of this overview information.
[8] Draper Laboratories in Cambridge, Massachusetts, contracted to provide elements of the FTPP system.
[9] This is a recent update to the evolving architecture definition, which did not include much information regarding the function of the simulator.

failure of one of the other FCC's.

The NE's in the five FCC's are interconnected via high speed fiber optic links and synchronized with a distributed fault tolerant clock, creating a network to support synchronous data communications among the processors in the five FCC's. The NE's receive input data from the processors, detect and mask data faults with a voting mechanism, and synchronously distribute congruent (voted) output data with error status information to other processors in the network.

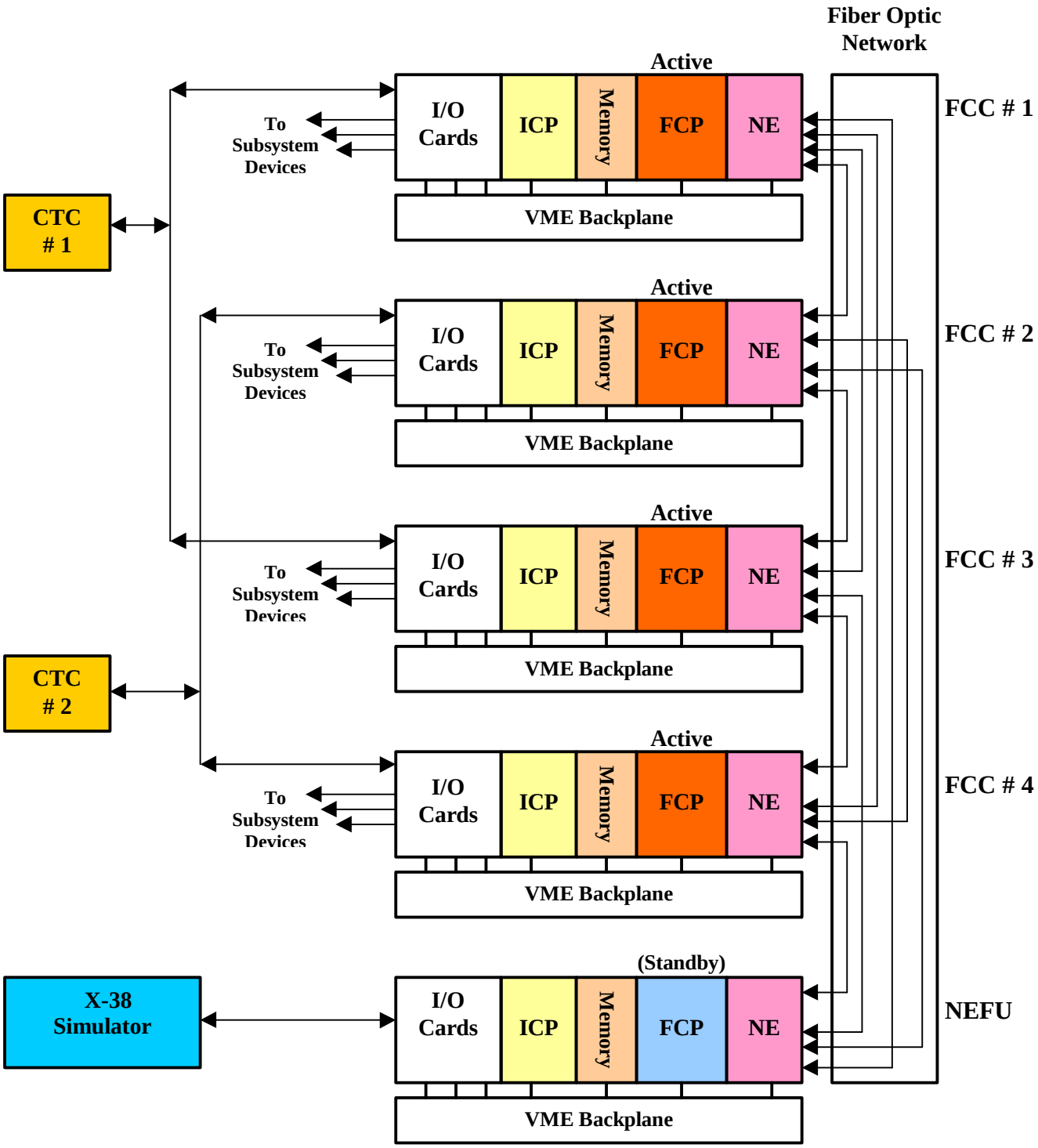


Figure 1 – Major Flight Critical Computer Hardware Components

The network supports data communications between *virtual groups (or virtual processors).* The NE's maintain configuration data, grouping functionally identical processors into virtual groups. The five redundant FCP's comprise a single FCP virtual group, creating a *virtual FCP*. Each of the five ICP's is identified in the configuration as a unique virtual group, since processing in the ICP's is dependent upon the asymmetric I/O device configurations supported by the ICP's. Communications among the processors is further characterized as follows in the documentation:

> "Communication between the five ICPs and the virtual FCP has the simple characteristics of a bus. The ordering and timing of inter-processor messages is guaranteed to have bus semantics, even in the presence of an arbitrary fault. These guarantees are collectively called the Byzantine Resilient Virtual Circuit (BRVC) abstraction."[10]

Figure 2 is a stylized version of an illustration of the *Virtual Architecture* extracted from the documentation which represents the network as a virtual bus. The following comment taken from the documentation elaborates on the concept of a virtual FCP from a software perspective:

> "The FCP redundancy is transparent to the applications programmer. As far as the programmer is concerned, the group of four processors appears as a virtual single processor with multiple*…[I/O interfaces]*. The fifth FCP acts as a standby processor which can become an active member of the voting group when another FCR fails."

Note that although all NE's participate in the network communications (including, particularly, voting), the FCP in the NEFU is designed to operate in standby mode, to be activated only in the event of an FCR failure (i.e., failure of an active FCC).

[10] Section 2.3, p.8, FTPP SRS [6].

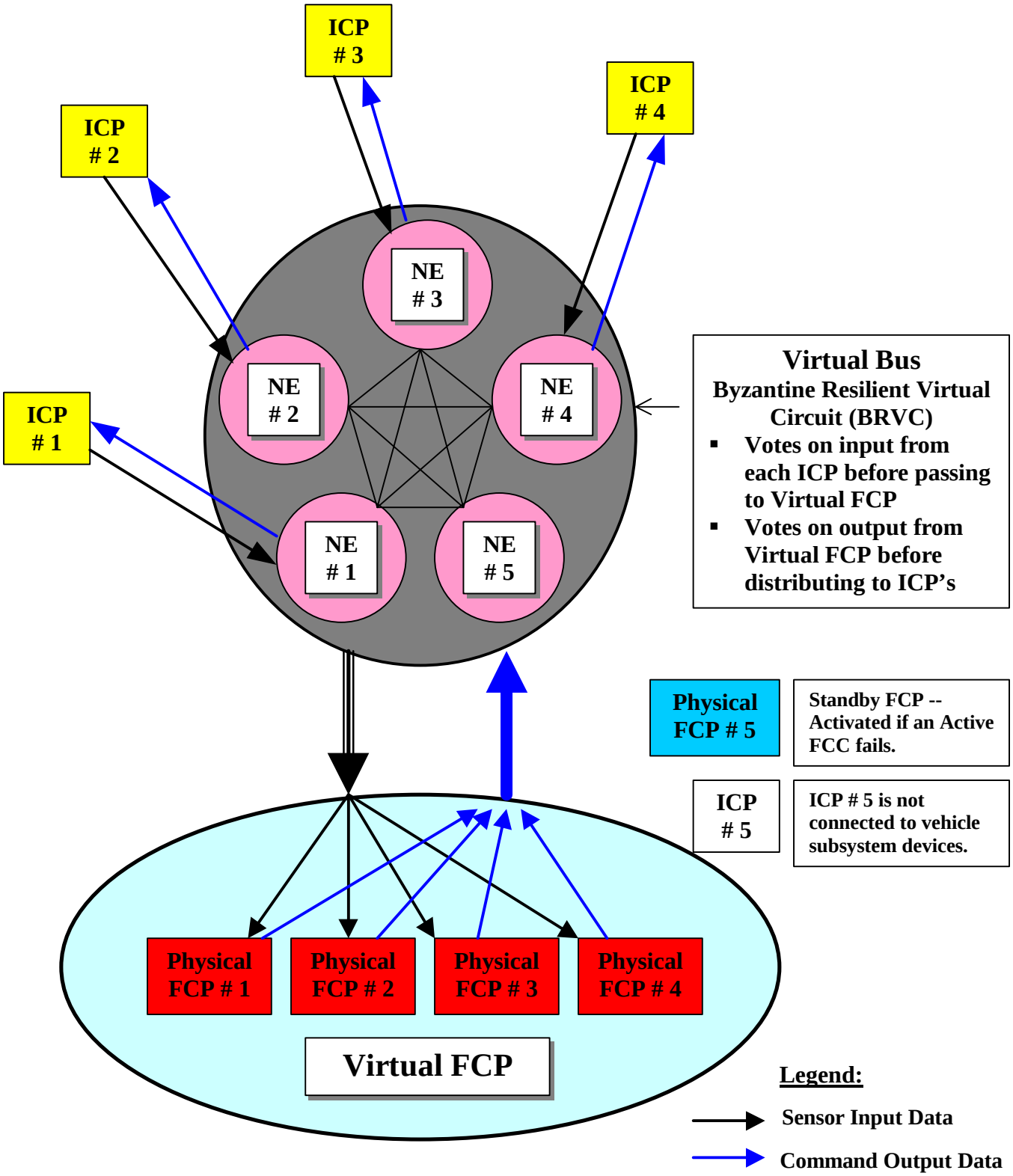
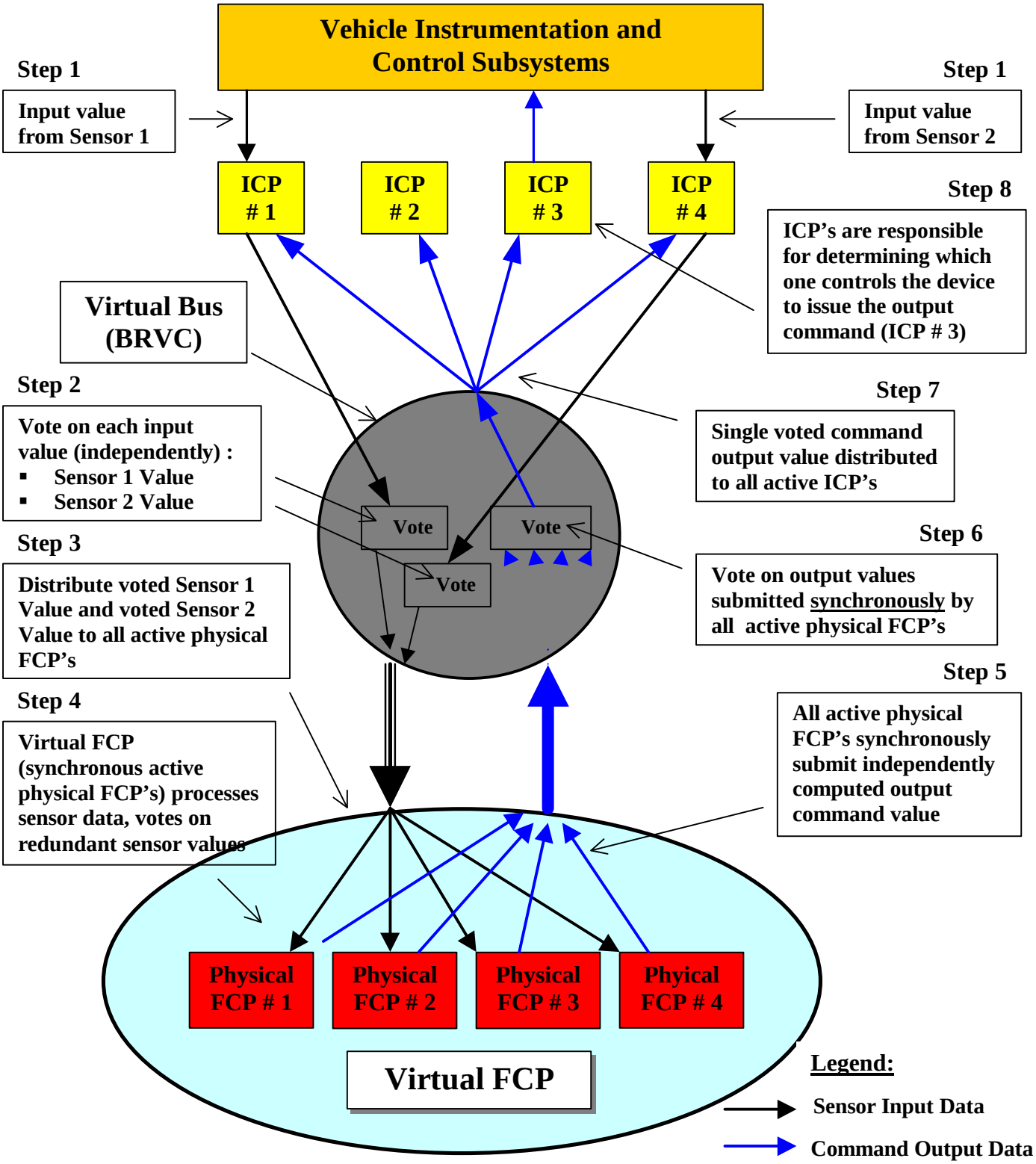


Figure 2 – Flight Critical Computer Virtual Architecture

Figure 3 depicts the data flow through the system to summarize the overview of the architecture. A sensor value is read from a vehicle instrumentation subsystem by the ICP, passed through the BRVC and delivered to the virtual FCP. The BRVC distributes the voted sensor value to all physical FCP's with the guarantee that all processors receive congruent values. The FCP applications receive the sensor value and process it. Note that values from redundant sensors are received by different ICP's and delivered to the virtual FCP independently. The FCP applications are responsible for managing the data from redundant sensors, for example, with some voting mechanism or other algorithm. Responding to the processed sensor value, identical applications in all physical FCP's generate a command value to control some affected vehicle subsystem. The physical FCP's synchronously present their independently generated command values to the BRVC, which votes on the values, generating a single command value as the output of the virtual FCP. The BRVC broadcasts the voted command value to all ICP's and echoes the value and the voting results to the physical FCP's (to support fault management and logging). The ICP software is responsible for determining which ICP controls the affected vehicle subsystem. This ICP then writes the command value to the vehicle subsystem to effect the required action.



NOTE: Data flow begins with Step 1, top right, and flows counterclockwise through Step 8.

Figure 3 – Flight Critical Computer Data Flow

The architectural design objectives with regard to fault tolerance are stated as follows:

> "The FCC *[i.e., the flight critical computer system]* is designed to tolerate two permanent, sequential processor failures. It is also designed to discriminate between transient faults, such as those caused by Single Event Upsets, and permanent faults and to recover FCC *[i.e., flight critical computer system]* hardware resources affected by transient faults."[11]

This brief overview of the hardware architecture, in addition to identifying major system components and key architectural concepts, suggests the obvious conclusion: a high level of complexity can be expected in the software.

3.2 Software Architecture

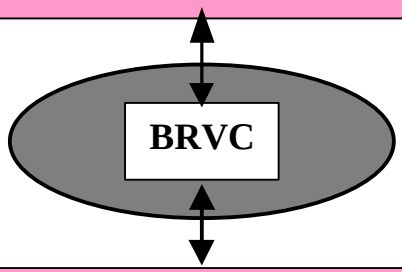
This overview focuses on those aspects of the X-38 201 software architecture that concern fault tolerance and fault evasion methods as previously defined. Further, the focus is primarily on the *software environment* in which the flight critical applications will operate, including methods to be implemented in system software or common applications software. Methods unique to particular applications, including, for example, algorithms unique to a particular application domain, are not considered except in a general sense. Moreover, methods that extend beyond the software system boundary, for example, vehicle safing methods implemented in hardware or manually, are not considered. Thus, the focus of this discussion will be on the system software, the applications structure, and applications interfaces to the system software, primarily in the FCP and ICP, with only some mention of the software in the Command and Telemetry Computers (CTC's).[12]

Conceptually, there are several layers of software in the FCP and ICP, as depicted in Figure 4 (which is a stylized version of an illustration in the reviewed documentation).[13] The three lowest level layers are common to the FCP and the ICP. At the very lowest level (really at the hardware level), is microcode in the NE[14] that implements the network interface functions (i.e., the previously defined BRVC). These functions include a voting algorithm. The second layer of software is a COTS real-time embedded operating system, which is responsible for controlling and managing the computer resources and supporting the hosted applications with standard operating system services. The third layer of software extends the COTS operating system to support the unique hardware and software architecture. This layer includes custom system software, for example, to handle unique device interfaces, and a set of Fault Tolerant System Services (FTSS).[15]

[11] Section 2.2, p.5, FTPP SRS [6].
[12] The CTC's are not considered flight critical.
[13] See Figure 2.4.1, FCC Software Architecture, FTPP SRS [6].
[14] The NE microcode and the FTSS software, including the API, are to be provided by Draper Laboratories.
[15] See footnote 14 above.

FCP Software Layers

<table>
<tr><td rowspan="2">Applications</td><td>Application Procedures</td><td colspan="3">Application Procedures
▪ Provided for each vehicle subsystem application area
▪ Structural template with Main procedure and generic set of called procedures
▪ Categorized by data rate and flight criticality
▪ Additional system applications services supported with procedures</td></tr>
<tr><td>Application Tasks</td><td>Cyclic Master Tasks
▪ Control application execution flow
▪ 5 categories –
▪ 50 Hz FC
▪ 50 Hz NFC
▪ 10 Hz FC and NFC
▪ 1 Hz NFC</td><td>Applications Services Tasks
▪ Mission manager initialization
▪ 50 Hz sensor input task
▪ 50 Hz effector output
▪ Inter-rate group data read
▪ Inter-rate group data write
▪ 50 Hz FTSS Fault Detection and Recovery task</td><td>Shared Memory Pools (Partitioned by application)</td></tr>
<tr><td rowspan="2">Extensions to the COTS OS</td><td colspan="4">FTSS Applications Programming Interface (API)</td></tr>
<tr><td colspan="4">Fault Tolerant System Services (FTSS)
(FCP Subset)
Scheduling Services — Timer Services
Memory Management Services — Hardware Fault Detection & Recovery
Interprocessor Communication Services — FCP-specific I/O Handling Services (CTC Interface)</td></tr>
<tr><td>COTS OS</td><td colspan="4">COTS Realtime Operating System Services
(FCP Subset)</td></tr>
<tr><td>(Hardware)</td><td colspan="4">NE Microcode</td></tr>
</table>



ICP Software Layers

<table>
<tr><td>(Hardware)</td><td colspan="2">NE Microcode</td></tr>
<tr><td>COTS OS</td><td colspan="2">COTS Realtime Operating System Services
(ICP Subset)</td></tr>
<tr><td rowspan="2">Extensions to the COTS OS</td><td>FTSS
(ICP Subset)
Interprocessor Communication Services
Fault Detection & Recovery</td><td rowspan="2">Custom System Software
(ICP Unique)
Support array of I/O cards (FCC's only)
Support X-38 Simulator interface (NEFU only)</td></tr>
<tr><td>FTSS Applications Programming Interface (API)</td></tr>
<tr><td>Applications</td><td colspan="2">Vehicle Instrumentation and Control I/O Management Tasks</td></tr>
</table>

Figure 4 – FCP and ICP Software Layers

The FTSS software includes, particularly, interprocessor communications services, and hardware fault detection, identification, and recovery (FDIR) services, which create the FTPP computing environment. Some FTSS services are unique to the FCP. These include intertask communications services and timer services to support a real-time clock interface and to provide special timers needed by the FCP applications. Among the FTSS services are "sanitized" replacements for certain standard COTS operating system services. These include task scheduling services and dynamic memory allocation services. They are designed to insulate the FCP applications from operating system interface mechanisms, which are presumably complex and potentially disruptive to the system to create a simplified operating systems environment for the applications software. An integral part of the FTSS software, and included in the layer of operating system extensions, is an applications programming interface (API), standard procedures and calling parameter definitions to be used by the interfacing applications software.

Note that the COTS operating system configuration and the FTSS configuration, in addition to the processor-unique custom system software, will differ in the FCP and ICP. Though derived from common software baselines, the configurations will be tailored to the unique functional requirements of the two processors. For example, some FTSS services will not be available in both processors.

The upper layer of software in the ICP is basically a set of I/O tasks. The input tasks read sensor data from the instrumentation hardware and write it to the virtual FCP via the BRVC. The output tasks read commands via the BRVC from the virtual FCP and write them to the appropriate subsystem control devices via the I/O interface cards in the FCC's. The ICP software is also responsible for managing the I/O device configuration, which is unique to each ICP. Though the software configurations are to be identical across all ICP's, the unique I/O device configurations will require logic in the software to control the I/O processing. Note that each ICP will be assigned a unique ID, available in FCC configuration information when the ICP is initialized. The ICP software will use this ID to determine the unique device configuration, making management and control of it possible. Also note that the I/O tasks will interface directly with the COTS operating system services and to the underlying layer of system extensions, which include the FTSS software tailored for the ICP.

In the upper layer of software in the FCP are the application procedures which process instrumentation sensor information and generate commands that are written to the vehicle subsystems. Procedures are provided for each application. The procedures are classified into three rate groups, which correspond to the frequency at which they are called and the time criticality of the data processed. The three rate groups are 50 Hz (every 20 ms), 10 Hz (every 100 ms) and 1 Hz (once per second). For the 50 Hz and 10 Hz rate groups, the procedures are also classified as flight critical (FC) or non-flight critical (NFC), with all 1 Hz procedures classified as NFC. For a particular application there can be procedures in more than one rate group and there can be both FC and NFC procedures. For example, for the guidance, navigation, and control application, there are 50 Hz FC, 10 Hz FC, and 1 Hz NFC procedures.

Table 2 identifies the major applications to be represented by individual Software Requirements Specifications.[16]

[16] This list does not include the FTPP, FCP, ICP, and CTC SRS's. See the list of specifications in the Software Verification and Validation Plan [8].

Table 2 – Major FCP Applications

Application Subsystem
Systems Management[1]
Communications and Tracking (C&T)
Electrical Power Distribution and Control (EPDC)
Propulsion
Attitude Control
Electromechanical Actuator
Environmental Control and Life Support[2]
Pyrotechnics
Parafoil
Flush Air Data Systems (FADS)
Space Integrated Global Positioning System (GPS) Inertial Navigation System (INS) (SIGI)
Guidance, Navigation, and Control
Orphan Software[3]

[1] Includes Vehicle (subsystems) and Mission (flight phase) management.
[2] This is an example of a prototype of a safety critical CRV system.
[3] Miscellaneous software.

Underlying (and surrounding) the layer of application procedures is a set of tasks structured by rate group and flight criticality. These tasks include "cyclic master" tasks that drive the application procedures, imposing a standard structure on the application software, establishing a standard processing flow, and prescribing standard application interface mechanisms, including the use of shared memory pools. Other tasks in this layer interface with the underlying system level software, including the FTSS software, on behalf of the application software. For example, 50 Hz sensor reader tasks are provided to read 10 Hz FC, 10 Hz NFC, and 1 Hz data from the ICP's and maintain "sensor memory pools" for lower rate tasks. Table 3 simply identifies the tasks, including the common system services tasks, and generic procedures[17] defined by the structure. See the description of the software architecture in [5] for elaboration. Observe that the indentation in the table suggests the structural layering of the software.

[17] They are generic in the sense that they are instantiated for the various applications, so for example, there are numerous instantiations of the SOP procedure, and across rate groups as shown in the table.

Table 3 – FCP Tasks and Generic Procedures Tasks[1]

Generic or Common Component	Task or (Generic) Procedure	50 Hz FC	50 Hz NFC	10 Hz FC	10 Hz NFC	1 Hz NFC
Rate Group Cyclic Master Task	Task	X	X	X	X	X
Initialization	Procedure	X	X	X	X	X
Executive	Procedure	X	X	X	X	X
Checkout	Procedure	X	X	X	X	X
Sensor Reader	Procedure	X	X			
Effector Writer	Procedure	X				
Data Collector[2]	Procedure		X		X	X
Pipe Reader[3]	Procedure		X	X	X	X
Application Main (Executive) Procedure[4]	Procedure	X	X	X	X	X
Subsystem Command Processing[5]	Procedure	X	X	X	X	X
Subsystem Operating Procedure (SOP)[6]	Procedure	X	X	X	X	X
Application Calculations[7]	Procedure	X	X	X	X	X
Effector Command Generator (reverse SOP)	Procedure	X		X	X	X
Pipe Writer[8]	Procedure			X	X	X
FTSS Fault Detection, Isolation, Recovery (FDIR)	Task	X				
Sensor Reader for 10 Hz FC Task	Task	X				
Effector Writer for 10 Hz FC Task	Task	X				
Sensor Reader for 10 Hz NFC Task	Task	X				
Effector Writer for 10 Hz NFC Task	Task	X				
Sensor Reader for 1 Hz NFC Task	Task	X				
Effector Writer for 1 Hz NFC Task	Task	X				
Mission Manager	Procedure		X			
Vehicle Manager	Procedure		X			
Remote Command Receptor	Procedure				X	
Telemetry Data Logger	Procedure				X	

[1] This table is based primarily on Table 2.1.1 in the SADD [5]. Note that the original table did not identify a task level Exception handler mentioned elsewhere in the documentation.
[2] The 50 Hz data collector reads lower rate sensor data and passes it to lower rate group applications in memory pools.
[3] Data communications between rate groups are accomplished using “pipes” (one method).
[4] Note that the generic procedures listed below the Application Main procedure are listed in the order of the processing flow suggested by the structural template defined by the software architecture.
[5] Each application is responsible for processing remote commands sent to a particular subsystem; these are provided to the application on a task level command queue.
[6] For each application, the SOP is responsible for sensor data conversion (raw values to engineering units), sensor redundancy management (voting), and sensor fault detection and recovery.
[7] These are the applications procedures that perform calculations using the sensor data.
[8] See note 3.

3.3 Software Requirements

Following is a summary of some of the major software requirements pertinent to the discussion in this report. These are particularly relevant when considering fault tolerance and the reliability of the software.

1. The redundant copies of the software in the FCP's will execute synchronously to effect a virtual FCP. Data exchanges with other processors must be completed within a configuration defined (and enforced) time skew that enforces this synchronization requirement. Time-out errors associated with this requirement include "Outbut Buffer Not Empty (OBNE)", meaning that an FCP failed to keep in sync with a majority of FCP's when transmitting data packets, and "Input Buffer Not Full (IBNF)", meaning that an FCP failed to keep in sync with a majority of the FCP's when receiving data packets. [18]

2. Functionally identical processors will be loaded with identical software. Software in the FCP's will be redundant and software in the ICP's will be redundant. [19] Further, there is a stated assumption that the two CTC's will have identical software loads.

3. The FCP applications software will execute in one of three rate groups: 50 Hz, 10 Hz, and 1 Hz, as previously described, with execution time bounded (i.e., enforced) by the task scheduler.[20] This is required to satisfy the data transport requirements of the system. Failure of a task to stay within the time budget will trigger an exception. Observe that these rate groups correspond to cycles of minor frames of data (every 20 ms), medium frames of data (every 100 ms), and major frames of data (every second). Further, FC and NFC processing is bounded separately, that is, there are FC and NFC data cycles. Also processing will differ for some minor frames, that is, there are asynchronous 50 Hz data cycles, so that, for example, processing for minor frames 0, 5, 10, etc., will be different from processing for minor frames 1, 6, 11, etc. This obviously adds a degree of complexity throughout the software.

4. FCP applications software will not be permitted to interface directly with the COTS operating system software, as stated in the following quotation:

   > " Although [the COTS operating system] is running on all of the FCP computers, application developers will not be able to use most [COTS operating system] calls. Instead, Draper Labs will provide an Application Programmers Interface (API) which contains sanitized versions of [the COTS operating system] calls. This is necessary to prevent applications from using blocking calls, which would adversely affect synchronization between the four FCPs."[21]

   Note that this restriction does not apply to applications software (i.e., the I/O tasks) in the ICP:

   > "The user I/O tasks can invoke [COTS operating system] services without any constraints."[22]

5. Memory pools used by the FCP applications software will be organized into congruent data (identical across processors) and non-congruent data. The FTSS software will attempt to maintain congruence with various "memory realignment" mechanisms.[23]

6. The architecture prescribes handling faults at the lowest level in the software where the fault can be detected and dealt with effectively. Thus, a major function of the FTSS software, which is an extension of the COTS operating system software, is fault detection and recovery. This approach is suggested in the following requirement:

[18] See requirements 3.2.2.6.6, 3.3.4.16, and 3.3.11.1 through 3.3.11.3 in the FTPP SRS [6].
[19] See requirements 3.1.4 and 3.1.5 in the FCP SRS and requirement 4.2.1.1 in the ICP SRS [9].
[20] See numerous requirements, 3.3.4.1, 3.3.4.5, and 3.3.4.12 through 3.3.4.17, in the FTPP SRS [6].
[21] See Section 1.0 of the SADD [5].
[22] See Section 2.4 in the FTPP SRS [6].
[23] See requirements 3.3.8.14 and 3.3.9.1 in the FTPP SRS [6].

"Computer redundancy and fault tolerance shall … be transparent to the application programs."[24]

Note, however, that applications will participate in fault recovery as exemplified in the following requirement:

"Upon occurrence of an exception, the FCP shall…log the error, …transfer control to a user specified exception handling routine if one is provided, and … restart the offending task at the location specified by the user at task creation.[25]

Note that beyond this statement, the requirements do not prescribe or recommend the processing to be performed in response to various exceptions, for example, in a default or template application exception handling routine.

7. "The software shall (3.3.1.2) be written in the C programming language…"[26]

8. The previously summarized system transport requirements for flight critical data, the FCP synchronization requirements, and the cyclic data processing requirements, impose a number of timing constraints (i.e., lower level requirements) on the applications and the FTSS software. These are exemplified in the following sample requirements (which are footnoted to provide additional explanation):

   "The FC output schedule must be as follows:
   a. 50 Hz FC outputs in each minor frame before expiration of the 6 millisecond time period in the ICP.
   b. 10 Hz FC outputs in minor frame 2 of each 10 Hz medium frame before expiration of the 14 millisecond time period in the ICP."[27]

   "NASA application code must use FTSS Communications Sevices for all FCP inter-rate group communications and for all communications between the FCP virtual group and the ICPs."[28]

   "For the highest rate group tasks (i.e., tasks that can not be preempted), Immediate Message Services shall … also be provided [in the FTSS software]."[29]

9. The software will be required to adapt to different processing requirements associated with the mission phases[30] and the possible transient or permanent failure of system components that determine the operational (and recovery) state of the system.

3.4 Fault Handling Methods Used

Table 4 lists all fault tolerance and fault evasion methods shown previously in Table 1, indicating whether these methods are applied in the X-38 software and whether they are applied to handle software or hardware faults. Table 4 also shows where in the software these methods are applied in terms of the software layers defined in the software overview. Consideration of the fault prevention techniques listed in Table 1 is deferred to the discussion of the development approach.

Note that the approach to handling exceptions generally excludes the use of exception handlers in [the COTS operating system], as stated in the following quotation (which presumably extends to the ICP software, as well):

---

[24] See requirement 3.3.13.1 in the FTPP SRS [6].
[25] See requirements 3.3.12.1, 3.3.12.2, and 3.3.13.3 in the FTPP SRS [6].
[26] See requirement 3.3.1.2 in the FTPP SRS [6].
[27] See requirement 3.4.1.2 in the FTPP SRS [6].
[28] The rationale for this requirement is to prevent timing disruptions. See section 3.3.6 in the FTPP SRS [6].
[29] In permitting immediate data exchanges, the software in the FCP's must be very tightly synchronized – i.e., synchronized at the data exchange points. See requirement 3.3.6.11 in the FTPP SRS [6].
[30] The FTSS software includes a template for the Mission Manager task, the applications software responsible for transitioning the system through the phases of the mission. See section 3.3.3 in the FTPP SRS [6].

> "Exceptions may be raised by FCP hardware components…These may be in response to [single event upsets], intermittent or permanent hardware faults, or residual software errors…"[31]

To reiterate, the general policy for handling exceptions in the software in response to faults (hardware and software), regardless of the number of FCC's affected is to do the following (in the FTSS software):

> "…log the error,…transfer control to a user specified exception handling routine if one is provided, and… restart the offending task at the location specified by the user at task creation." [32]

Given this approach, the COTS operating system is excluded from Table 4 and from consideration in the remaining discussion (with the assumption that it works correctly interfacing with the X-38 software, including the FTSS software).

[31] See section 3.3.12 in the FTPP SRS [6].
[32] See requirements 3.3.12.1 through 3.3.12.3 in the FTPP SRS [6].

Table 4 -- Fault Handling Methods Used in the X-38 201 Software[1]

	Purpose		Use in the X-38 Software				
Method	**Fault Tolerance**	**Fault Evasion**	**NE Microcode**	**I/O Extensions**	**FTSS**	**Application Tasks**	**Application Procedures[2]**
N-versioning[3]	X						
Partitioning[4]	X				H,S[5]	H,S[6]	S[7]
Voting	X		H,S[8]		H,S[9]	H,S[10]	H,S[11]
Retry	X			H[12]	H,S[13]	H,S[14]	
Recovery Block	X					H,S[15]	
Distributed Recovery Block	X				H,S[16]		
Heartbeat Messages	X				H,S[17]		
Watchdog Timers	X			H[18]	H,S[19]		
Input Analysis		X					No
Output Analysis		X					No
Consistency Checks		X			H,S[20]		No

[1] "H" and "S" indicate that the method deals with hardware faults and software faults, respectively.
[2] The documentation that was reviewed did not include requirements specifications for application procedures, i.e., the upper layer of FCP software.
[3] N-versioning is not used at any level.
[4] Partitioning as a fault prevention method is reflected in the software architecture (i.e., in the layers of software previously discussed) in part, by simplifying the software, making it, from a human factors perspective, more manageable.
[5] The FTSS software implements memory protection mechanisms for applications tasks, and mechanisms for maintaining congruent memory pools for applications (i.e., implementing, in effect, "virtual shared memory."). This isolates FCC hardware memory faults and software data value faults to the FCC. The FTSS software also implements mechanisms in the task scheduler to measure and limit processor consumption by tasks. Exceeding a time quota could be caused by software faults (typically) or hardware faults.
[6] Included in the application task structure, are mechanisms for partitioning shared memory by rate group and by major application. This has the effect of containing hardware and software faults (such as memory exceptions) to specific tasks where they can be dealt with by the individual task exception handlers.
[7] Partitioning the software into procedures, that is, modularizing the software, is a fault prevention technique, as applied in the X-38 software. The modular structuring can also be used with other comparison techniques to contain, detect, and attempt to recover from data value faults occurring in the procedure.
[8] The NE microcode implements a bit-by-bit comparison of input values in data exchanges among processors (e.g., from an ICP to the virtual FCP). The voting algorithm attempts to select a majority value. In the case of a tie, the bit-by-bit comparison logic favors a "1" value, meaning that *the voted (output) value can differ from any of the values used in the comparison.* Thus, it would appear that the voting algorithm *can create a data value fault.* The documentation did not explain how the software would handle the apparent faulty data values, nor did it provide the rationale for choosing this voting algorithm.
[9] The FTSS software is responsible for managing FCC hardware redundancy with various voting mechanisms to determine the health and status of FCC hardware components, including the NE's, the FCP's, and the ICP's. Failing components can be "voted out" of the configuration.
[10] In the FCP voting is used in the application tasks to manage redundant vehicle subsystem hardware by voting on health and status information. In the FCP voting is also used to examine command input from the redundant CTC's to determine the source of remote commands to the vehicle subsystems.
[11] Application procedures are responsible for voting on the redundant sensor values used in processing. Typically, this voting will mask hardware faults, except in the case of a sensor value corrupted by previous software processing.
[12] Retry is commonly used in I/O interface software.

[13] When attempting to recover from transient faults, the FTSS software will attempt to reinitialize a failing component (NE, FCP, or ICP); repeated unsuccessful retry attempts constitute a permanent failure of the component (and the FCC).

[14] The vehicle subsystem manager software in the virtual FCP will work cooperatively with the FTSS recovery software to retry a failing physical component (e.g., an FCP or NE). In attempting to determine if a component failure is transient or permanent, the FTSS software will attempt to reset (via command) the failed component. In the event that the reset is unsuccessful, the FTSS software will notify the vehicle manager software, requesting that power to the component be recycled in an attempt to reset a presumed "latchup" state in the component. Note that overall responsibility for controlling power is allocated to the vehicle manager software.

[15] The architecture prescribes a recovery block mechanism to be used by FCP applications to restore shared memory pools as part of FCP recovery processing following a fault. The architecture also *suggests* the use of application-specific recovery blocks by applications tasks in their exception handling procedures, though the details of this are left to the applications design.

[16] The FTSS memory management services in the FCP are responsible for "realigning" memory when attempting to recover from a transient fault and restore the faulting processor to the configuration. Realigning memory means making data in the processor's congruent memory pools consistent (i.e., identical) with the redundant copies in the non-faulty processors. In this manner, the redundant copies can be considered distributed recovery blocks.

[17] Heartbeat messages are used in the FTSS software to detect the possible failure of other processors in the configuration.

[18] The use of watchdog timers, a standard mechanism, to monitor the progress of I/O operations is assumed in this case.

[19] Watchdog timers are used in the FTSS software to detect the possible failure of I/O operations, including data exchanges among processors. They are also used by the FTSS scheduling services to detect task overrun, the condition when a task exceeds its allocated processor budget.

[20] The FTSS software is required to periodically "scrub" memory in the FCP's. This involves periodically verifying that congruent memory is consistent across all processors and that it is free of hardware errors within each FCC.

The following observations can be made from examining Table 4, including the footnotes, which elaborate on how the methods are applied:

1. Consistent with one of the findings in the first report, a combination of methods is used in the X-38 201 software to handle faults. These include fault tolerance methods and fault evasion methods. As well, some methods identified in the first report are not applied.

2. Most of the fault tolerance and fault evasion methods applied in the X-38 201 software are *in theory* applicable to both hardware and software faults. However, the architecture focuses on detecting and recovering from hardware faults, as demonstrated by numerous requirements.

3. Classifying software faults broadly as either timing or data faults, the methods that address software faults focus on handling timing faults, which can be caused by either hardware or software. These include output-buffer-not-empty, input-buffer-not-full, and task (processor budget) overruns.

4. Most of the methods being applied to handle faults are implemented in the FTSS software (which is consistent with the architectural focus on hardware faults).

5. N-versioning, identified in the first report as being a cornerstone in software fault tolerant systems, is not used at any level. The use of N-versioning is precluded by the architecture, which requires replicated software in all functionally identical processors.

3.5 Challenges Posed by the Architecture

Following is a summary of at least some of the major challenges apparent in the X-38 201 architecture. These may be considered special problems and areas of risk in the area of software fault tolerance and dependability.

1. Central to the software architecture is the concept of a virtual FCP, the collection of five physical processors executing identical software in unison. Because the software is required to be identical and to execute synchronously, a software fault manifesting in one FCP can be expected to manifest in all other FCP's. Any software fault becomes a *common mode fault.* If one FCP fails as the result of a software fault, the virtual FCP (i.e. all FCP's) can be expected to fail.

   The requirement for identical software is extended to the ICP's (though there will be some processing differences among the ICP's to account for I/O configuration differences) and to the CTC's. This extends the potential problem of common mode software faults to each group of like processors.

   Thus, an implicit assumption in the architecture is *that the software will not fail*, meaning that methods used to achieve fault tolerance in the system are targeted at handling hardware faults.

2. The robustness of the COTS operating system is assumed; there is the implicit assumption that it will not fail. Though examining the dependability of the COTS operating system is beyond the scope of this assessment, the criteria and process for accepting the COTS software present a special challenge to the software development effort.

3. The FTSS software, which extends the COTS operating system and has primary responsibility for handling faults, is new software that is now being developed. Representing essentially a new version of the operating system, it is not a tried and proven (i.e., field tested) version. Modifications to correct errors and to adjust the API should be expected. Given the obvious complexity of the FTSS software and its central role in the software architecture, verification is a significant challenge. This is particularly evident in attempting to demonstrate the software's ability to handle various kinds of faults in various system states (i.e., processor configurations and modes of operation), and in testing the time critical communications services. As software developed under contract by Draper Laboratory, the process for accepting the FTSS software product, including acceptance testing, is also a significant challenge.

4. Implementation of the software in the C language presents special challenges to developing dependable software. In general, as a low-level language, C does not embody modern software engineering principles found in other languages (e.g., Ada). For example, it is not a strongly typed language, meaning that data type mismatches cannot be detected statically (by the C compiler). Further, there are features of the C language (including certain syntax and an orientation toward the use of pointers to access data structures) which can introduce logic errors, if not well understood and controlled. [33]

5. There are several challenges posed by the architecture to the requirements management process:

   a) Fault handling is distributed among software components at all levels (i.e., in all software layers). This creates the potential for interface problems. The challenge is to maintain consistency in these requirements, which collectively satisfy one of the major requirements of the system.
   b) Requirements for exception handling, which is allocated to applications tasks in the software architecture, must be explicitly defined in detail, and perhaps supported with a prescribed processing template that addresses the various exceptions that the software will be required to handle. Requirements defining an appropriate response to exceptions triggered by software faults pose a unique challenge. Since a software fault is a common mode fault, an exception triggered in one processor would be expected to replicate across all like processors. The challenge is how to respond to an exception triggered by a software fault, particularly when the exception could also be triggered by a hardware fault (e.g., a memory exception error).

6. The use of shared memory pools poses at least two challenges:

   a) Shared memory pools are inherently vulnerable to corruption by errant software. Protecting it , with memory protection hardware features, by partitioning, and by other means, requires special consideration.
   b) Managing the configuration of the shared memory pools is a challenge to the software configuration management process.

4 Assessment of Software Development Approach

4.1 Characteristics of the Development Approach

As stated in the reviewed documentation, "The basic development philosophy of the X-38 project is a rapid prototyping approach of 'design, build, test, redesign, rebuild, retest, etc.' in small increments."[34] The objective is "to demonstrate early end-to-end functionality," accomplishing testing in parallel with the development of the software to achieve this end. Noting that the documentation describing the development approach is preliminary, the following major points characterize the development approach:

General:

1. A primary objective of the X-38 prototyping effort is "…to make the X-38 201 Vehicle and its test *as CRV-like as possible*."[35] Reusability of the software, which would be consistent with this objective, is a goal that is not emphasized in the development approach.
2. According to estimates in the reviewed Software Development Plan, most of the software will be new code. Reuse of code from predecessor systems is minimal.
3. The FTSS software, a critical component in the system, as previously described, is being developed under contract by Draper Labs.
4. Regarding the application of standards, a noticeable omission in lists of applicable documents is the NASA Software Safety Standard (NASA-STD-8719.13A)[36]. Aside from X-38 201 safety

[33] See the book "Safer C"[2].
[34] See p.17 in the Software V&V Plan [8].
[35] See p.10 in the Software V&V Plan [8].
[36] See [12].

concerns (e.g., the pyrotechnic systems, the possible effects of losing control of the vehicle, its presence on the Shuttle as a payload), though it will be unmanned, software safety is a consideration in the context of prototyping the CRV.

Requirements and Design:

5. The approach to managing requirements involves producing and evolving a software requirements specification for major software subsystems – the FTPP (including the FTSS software), the FCP, the ICP, and each flight subsystem application. The approach also includes evolving interface control documents (ICD's) for major software interfaces. Collectively, these documents are the software specification.
6. The Software Architecture Definition Document and CRV Requirements documents serve as the high-level requirements specifications. Traceability from the software requirements specifications to a consolidated, higher level (coordinating) requirements specification is not provided.
7. Presumably, the "lessons learned" from the prototyping effort will be documented in the evolving requirements specifications and the as-built software design documentation. Apparently, prototyping objectives are not formally documented.
8. Although an "as-built" software design specification is planned following delivery of the system, no formal one will precede development of the software.

Code:

9. To reiterate, most of the software will be implemented in the C language. Beyond considering specific language strengths and weaknesses, the use of C assumes a procedure-oriented, rather than an object-oriented design paradigm.
10. The project has developed a "C Coding Standard."

Testing:

11. Draper Labs has responsibility for performing most of the testing of the FTSS software. Notable exceptions are system integration testing and what is identified as "stress" testing.
12. An array of external testing equipment will be used in testing, including a "…main test control and simulation system…to be provided by COTS vendors" and "…custom simulations…developed early on…to support unit testing and early integration testing."[37]
13. The (preliminary) Software Verification and Validation Plan identifies the risk "…that some desired testing simply will not be performed, leaving some uncertainty in those areas [*to be determined*] about proper functionality/performance."[38] This risk is attributed to schedule and staffing concerns.
14. Testing will include some level of mission scenario testing.

4.2 Software Development Challenges

Table 5 summarizes some of the major challenges confronting the X-38 development effort to build a dependable software system. The table identifies challenges in several key development process areas. It also summarizes the potential impact of the development approach (particularly, the impact of evolving the software requirements), on the development process.

[37] See section 5.1.5, p.25 in the Software V&V Plan [8].
[38] See section 5.1.9, p.27 in the Software V&V Plan [8].

Table 5 – Software Development Challenges

Development Process	Challenges to the Process	Potential Impact of the Approach on the Process
Requirements Management	1. Maintaining consistency in the requirements (and avoiding redundancy) across multiple evolving SRS's, particularly in the absence of a definitive, coordinating, high-level requirements specification. 2. Avoiding redundancy across multiple SRS's. 3. Maintaining a sufficient level of detail in the requirements as they evolve.	1. Unable to apply validation methods (e.g., formal methods) to evolving requirements specifications, which by definition, are incomplete (particularly early in the definition process). 2. Unable to apply fault evasion methods in the code in the absence of very detailed requirements pertaining to expected data values (e.g., expected sensor value ranges). 3. Unable to apply N-versioning methods with an incomplete, continually changing requirements specification (disregarding the impact of the architecture on an N-versioning approach).
Configuration Management	1. Multiple software configurations to manage, including: ▪ COTS OS (FCP and ICP versions) ▪ FTSS (FCP and ICP versions) ▪ CTC ▪ Subsystem Applications ▪ Shared global memory structures 2. Managing the global memory structures, including FCP congruent and incongruent memory to maintain consistency with the applications. 3. Managing multiple SRS's. 4. Maintaining traceability in software and test products to evolving requirements – i.e., maintaining functional consistency among products.	1. Frequent configuration changes in response to changing requirements.
Implementation	1. Avoiding the pitfalls of the C programming language. 2. Communicating the software design, including lessons learned, in the absence of a build-to design specification. 3. Managing timing budgets of applications tasks. 4. Debugging time critical applications, particularly in a multiprocessor environment.	1. Precludes potential benefits of other languages such as Ada or C++. 2. Precludes potential benefits of other design paradigms such as an object-oriented approach. 3. Precludes reusability of software.

Development Process	Challenges to the Process	Potential Impact of the Approach on the Process
Testing	1. Maintaining consistency of test products with evolving requirements. 2. Verifying and accepting the critical FTSS software. 3. Demonstrating fault tolerance of the system. 4. Verifying numerous timing requirements. 5. Demonstrating system functionality and performance in various modes (states) of system operation determined by flight phase and recovery configuration. 6. Mitigating the stated risk regarding limiting the scope of the testing to prioritized functions.	1. Unable to apply fault injection methods in the absence of very detailed and stable requirements. 2. Independence of test team is compromised by their necessary involvement in requirements definition. 3. Unable to develop integration level tests early on, and to maintain consistency in them.

4.3 Fault Prevention Methods Applied

Table 6 summarizes the fault prevention methods previously identified in Table 1 that are being applied in the X-38 201 software development effort. Note that this summary is based on a review of the preliminary project documentation pertaining to the software development approach and a cursory review of the COTS operating system vendor tools to be used in the development effort. For those methods where there was no information, the assumption is that the method is not being applied, as indicated in the footnotes.

Table 6 -- Fault Prevention Methods Applied in the X-38 201 Software Development Effort

Method	Purpose		Applied?
	Fault Avoidance	Fault Removal	
N-versioning	X		No[1]
Formal Methods	X		No[2]
State Modeling	X		No[3]
Reliability Modeling	X		No[4]
Fault Analysis	X		No[5]
Fault Injection		X	No[6]
Subsetting			
Design Paradigm	X		**Yes**[7]
Coding Standards	X		**Yes**[8]
Environment Standards	X		**Yes**[9]
Automated Tools			
Requirements Management	X		No[10]
Configuration Management	X		**Yes**[11]
Design	X		No[12]
Code Analysis		X	No[13]
Code Generation		X	No[14]
Debug		X	**Yes**[15]
Test Generation		X	**Yes**[16]
Test Effectiveness Analysis		X	**Yes**[17]

[1] N-versioning (with the possible exception of low level procedures) is precluded by the system architecture.
[2] Assumed.
[3] Assumed.
[4] Assumed.
[5] Assumed.
[6] Assumed.
[7] The architecture prescribes different COTS operating system and FTSS software configurations for the FCP and ICP.
[8] A "C Coding Standard" has been developed for the project.
[9] Several programming constraints have been imposed on the applications software. These generally pertain to the use of mechanisms available to the applications to interface with the COTS operating system, the FTSS software, and applications software in other rate groups. The objective of these constraints is to restrict and control the use of interface mechanisms that are potentially disruptive to the system.
[10] No requirements management-specific tool (other than Microsoft Word, the standard word processor) was identified.
[11] The COTS product Visual Source Safe is being used.
[12] Assumed. A possible exception is in the use of Labview to generate support displays external to the system.
[13] Assumed.
[14] Assumed. A possible exception is code generation in Labview, a COTS product, to support displays external to the system.
[15] Included in COTS OS development environment.
[16] Apparently some level of test generation is a capability of the external test and simulation system.
[17] Apparently some level of automated test effectiveness analysis is provided in the external test and simulation system. It is doubtful that this performs detailed analysis within the FCC, including, for example, test coverage analysis or complexity measurement of the code in the system.

5 Opportunities

5.1 N-versioning Not an Option

The system architecture precludes the use of an N-versioning approach by requiring the replication of the software (i.e., physical redundancy) in each group of functionally like processors (FCP, ICP, and CTC). While the system might tolerate a hardware fault that could cause the system to fail if not handled properly, it cannot tolerate a software fault that would have a similar effect. Though there may be certain kinds of software faults that the software can be programmed to handle, a fault that cannot be handled effectively can be expected to replicate across the group of processors, causing the entire group to fail. This is particularly true in the FCP's, the heart of the system, where the processing flow is essentially identical across processors. Conceptually, a software fault in a physical FCP is propagated to the virtual FCP.

With N-versioning essentially eliminated as an option, the emphasis naturally shifts to building a robust software system, one that is highly dependable and, so far as possible, unable to fail. Regardless of whether an N-versioning approach is used or not, the focus should be on fault prevention – on applying fault avoidance and fault removal methods in the first place. Multiple versions can all be defective. The fault prevention methods can be supplemented with fault handling methods, which can also facilitate removal of faults in testing.

5.2 Other Opportunities

Table 7 summarizes the opportunities identified for improving dependability in the X-38 software, given the architecture. The table also includes suggestions regarding how these opportunities might be extended to other software systems.

Table 7 – Opportunities for Improving Dependability

Process Area	X-38 Software	Beyond to Other Software
Approach	1. Emphasize reusability of the software as a development goal, consistent with the prototyping objective. 2. Review the applicability of various standards, particularly the NASA Software Safety standard, and apply as required. 3. Document the rationale for various major architectural, requirements, and design decisions, including the following: • choice of the C language • procedural rather than object-oriented design paradigm • choice of COTS software • voting algorithms 4. Identify and apply some fault analysis method (e.g. software fault tree analysis)[1] to classify faults to a level appropriate for supporting fault handling (including application exception handling) and fault injection.	1. Extend the emphasis on reusability to other software systems, to improve dependability (and reduce costs). 2. Reemphasize the application of software safety analysis methods, verifying that they can be practically applied -- that they are consistent with modern software development approaches, as simple as possible, and communicated to development organizations, for example, through work instructions.
Requirements Management	1. Identify and apply some automated requirements management tool to help maintain consistency in the evolving requirements. 2. Verify that fault detection and recovery requirements are specified to the level of detail necessary to support fault injection techniques and fault evasion methods (input analysis, output analysis, consistency checks), as well as testing.	1. Define a standard automated process for managing requirements.
Development	1. Refine the C Coding Standard by comparing it to other, similar standards, and incorporating the best methods among those surveyed. 2. Implement standard coding mechanisms to circumvent potential problems inherent to the language. 3. Identify and apply automated code analysis tools to enforce the coding standard. 4. Prescribe a standard exception handler (template) for applications tasks, identify the exceptions to be handled, and define the default processing for each exception.	1. Develop and maintain a definitive (global) C language standard supported with standard coding mechanisms and a standard set of code analysis tools, extending this to other languages used frequently in safety critical software.
Testing	1. Adopt a testing goal oriented toward "malicious" testing with regard to the fault handling – i.e., focus on trying to "break" the system to demonstrate its ability to tolerate faults. 2. Define and apply fault injection methodology to demonstrate the ability of the system to handle certain classes of faults (noting that this will depend on the results of some software fault analysis). 3. Apply appropriate test analysis tools to measure the effectiveness of testing (e.g., test coverage analysis).	1. Research the area of fault injection with the objective of defining a standard methodology that can provide some assurance measure of fault tolerance.

[1] The NASA Guidebook for Safety Critical Software [11] recommends the following on p.84:
"In systems where the cost of failure is high, special techniques or tools such as Fault Tree Analysis (FTA) need to be used to ensure safe operation." Also see Appendix B in [11].

6 Appendices

6.1 X-38 201 Software Documents Reviewed

The X-38 201 documentation listed in Table 8 was reviewed. Note that this list, particularly the version numbers, help to characterize the state of the software documentation at the time of the review.

Table 8 -- X-38 Software Project Documents Reviewed

Type[1]	Document Title	Version	Date
SRS	X-38 201 Software Requirements Specification, Command & Telemetry Subsystem Antenna Algorithms	Baseline, Version 1.0	05/13/99
SADD	X-38 201 Software Architecture Definition Document	Version 2.1	07/27/99
System Requirements	X-38 201 Fault Tolerant Parallel Processor Requirements	Revision 5.4	07/30/99
SRS	X-38 201 Software Requirements Specification, System Management	-	07/28/99
ICD	X-38 201 Software Requirements Specification, FCP Systems Management to FCP Applications ICD	-	06/31/99
SRS	Software Requirements Specification for the Instrumentation Control Processor (ICP) of the X-38 Vehicle 201 Flight Critical Computer (FCC) System	Version 1.0	02/25/99
SRS	X-38 Vehicle 201 Software Requirements Specification, Orphan Systems	Version 1.0	05/28/99
SRS	X-38 Vehicle 201 Software Requirements Specification, Pyrotechnics Subsystem	Preliminary Version 0.2	07/09/99
SRS	X-38 Vehicle 201 Software Requirements Specification, Electrical Power Distribution & Control (EPDC) Subsystem	Draft 2	04/01/99
SRS	Software Requirements Specification for the X-38 Vehicle 201 Command and Telemetry Computer (CTC) Software	Baseline, Version 1.2	07/99
ICD	X-38 Vehicle 201 Command and Telemetry Computer (CTC) to Ethernet Local Area Network Interface Control Document (ICD)	Draft	07/99
ICD	X-38 Vehicle 201 Command and Telemetry Computer (CTC) to Flight Critical Computer (FCC) Interface Control Document (ICD)	Draft	07/99
SRS	Software Requirements Specification (SRS) for the X-38 Vehicle 201 Environmental Control and Life Support (ECLS) System Software	Draft	12/98
SRS	X-38 Vehicle 201 Software Requirements Specification, Flight Critical Processor	Version 0.1	07/02/99
Slides	X-38 201 SRR/PDR Slide Presentation	-	02/26/99
SDP	Software Development Plan for the X-38 201 Vehicle	Revision B	05/28/99
Verification Plan	Software Verification and Validation Plan for the X-38 201 Vehicle, JSC 28717	Revision A	07/30/99
Coding Standard	C Coding Standard for the X-38 v201 Project, JSC 28658	Baseline	04/99
Design Specification	Application Programmer's Interface for the X-38 Fault Tolerant System Services		01/08/99

[1] SRS – Software Requirements Specification; SADD -- Software Architecture Definition Document; ICD – Interface Control Document

6.2 Cited References

[1] Heimerdinger, Walter L., and Weinstock, Charles B., A conceptual framework for system fault tolerance, CMU/SEI-92-TR-33, October 1992, Software Engineering Institute, Carnegie Mellon University.
[2] Hatton, Les, "Safer C", McGraw-Hill Book Company, 1995.
[3] Leveson, N.G., and Knight, J.C., (available on the Web), A reply to the criticisms of the Knight & Leveson Experiment.
[4] Software Verification and Validation Plan for the X-38 Vehicle, JSC-28717, Revision A, 07/30/99.
[5] X-38 201 Software Architecture Definition Document, Version 2.1, 07/27/99.
[6] X-38 201 Fault Tolerant Parallel Processor Requirements, Revision 5.4, 07/30/99.
[7] X-38 201 SRR/PDR Slide Presentation, 02/26/99.
[8] Software Verification and Validation Plan for the X-38 201 Vehicle, JSC 28717.
[9] Software Requirements Specification for the Instrumentation Control Processor (ICP) of the X-38 Vehicle 201 Flight Critical Computer (FCC) System, Version 1.0, 02/25/99.
[10] Weinstock, Charles B. and Gluch, David P., A perspective on the state of research in fault-tolerant systems, CMU/SEI-97-SR-008, Software Engineering Institute, Carnegie Mellon University, June 1997.
[11] NASA Guidebook for Safety Critical Software – Analysis and Development, NASA-GB-1740.13-96.
[12] NASA Software Safety Standard, NASA-STD-8719.13A.